Proof Support for IMP++ in HOL-OCL

Master Thesis

Lukas Brügger

Information Security
ETH Zürich

August 2, 2007
Overview

- There is a gap between verification on the specification level and on the implementation level.
- We close this gap by extending HOL-OCL . . .
- . . . with a programming language semantics based on a Hoare-calculus.
- All encodings are strongly typed
Motivation

Node

next: Node

Cnode

color: Boolean
Motivation

Node
   next: Node

Cnode
   color: Boolean

inv flip: self.color <> self.next.color
Motivation

Motivation for Integrating IMP++ in HOL-OCL

Node
next: Node

Cnode
color: Boolean

inv flip: self.color <> self.next.color

N1 := new Cnode();
N2 := new Cnode();
N1.set_color true;
N2.set_color false;
N1.set_next N2;
N2.set_next N1;
return N1;
Motivation

**Node**
- next: Node

**Cnode**
- color: Boolean

**inv flip: self.color <> self.next.color**

Node
Cnode
color: Boolean
next: Node
inv flip: self.color <> self.next.color
N1 N2
T F
Background: HOL-OCL

HOL-OCL is an interactive proof environment for UML/OCL.

It provides:

- A datatype package for OO data structures
- A machine-checked semantics for OCL
- A proof calculi for a three-valued logic over path expressions
- A framework for analyzing OO specifications

Type constructor $\tau \, up$: assigns to each type $\tau$ a type lifted by $\bot$. 

Missing: Programming language semantics. Therefore no program verification.
Background: HOL-OCL

HOL-OCL is an interactive proof environment for UML/OCL.

It provides:

- A datatype package for OO data structures
- A machine-checked semantics for OCL
- A proof calculi for a three-valued logic over path expressions
- A framework for analyzing OO specifications

Type constructor $\tau_{up}$: assigns to each type $\tau$ a type lifted by $\bot$.

**Missing:** Programming language semantics. Therefore no program verification.
Background: Hoare Logic

Method for analysing programs with a small-step semantics. Main construct is a Hoare Triple:

\[ \models \{ P \} \ c \ \{ Q \} \]

“if $P$ holds for some state $s$ and $c$ terminates and reaches state $t$, then $Q$ must hold for $t$”

- $P, Q$ are assertions. Modelled as set of states
- Denotational Semantics used for relating states and commands
- Isabelle provides IMP: a Hoare calculus for an imperative language
Content

Introduction

Background

IMP++

Integrating IMP++ in HOL-OCL

Conclusion
**IMP++**

**Motivation:** Extend IMP by core features of object-orientation (Java subset)

Deep embedding: Syntax is introduced as datatype definition:

```plaintext
datatype
  α  com =
    SKIP
  |  Cmd  α  cmd
  |  Semi  α  com  α  com  (_ ; _)
  |  Cond  α  bexp  α  com  α  com  ( IF _ THEN _ ELSE _ )
  |  While  α  bexp  α  com  ( WHILE _ DO _ )
```

**types**

```plaintext
α  bexp = α  state  =>  bool  up
α  cmd = α  state  =>  α  state  up
```
Hoare Logic for IMP++

Fairly standard, but with support for undefinedness.

States can always be undefined. Constant $err$ denotes the error state. Used for modelling exceptions.

**types** $\alpha$ assn $=$ $\alpha$ state up $=>$ bool

**constdefs**

$\text{hoare\_valid} :: [\alpha \ assn, \alpha \ com, \alpha \ assn] => bool$

$\models \{P\} c \{Q\} \equiv \forall s \ t. \ (s, t) \in C(c) \rightarrow P \ s \rightarrow Q \ t$
Content

Introduction

Background

IMP++

Integrating IMP++ in HOL-OCL

Conclusion
The HOL-OCL/IMP++ Architecture

Level 2

Level 1 OCL

Level 1 IMP++

Level 0
The HOL-OCL/IMP++ Architecture

Object Store
- Universe and class types
- Getters for the attributes

Can be reused for IMP++, extended with the Setters.
The HOL-OCL/IMP++ Architecture

Level 1:
- Lifting over the context of any semantic function
- Explicit dealing with definedness and strictness
- HOL-OCL provides types and operators to automate this
The HOL-OCL/IMP++ Architecture

Level 1:
- The context of HOL-OCL is a pair of state (pre/post)
- The context of IMP++ is one state
→ Definitions and lemmas can’t be reused
The HOL-OCL/IMP++ Architecture

Level 2

Level 1 OCL  Level 1 IMP++

Level 0

Level 2 adds support for preconditions, postconditions and invariants.

Here we can relate the Hoare verification with the specification.
The commands of our language are state transitions. The state is a mapping from oids to the Universe with the following constraints:

- NULL must not be a valid reference
- All elements in the range of the state must be defined
- There is a one-to-one correspondence between objects and their oid (oid stored as part of the object)
State

The state is a mapping from oids to the Universe with the following constraints:

```constdefs correct_state :: "(oid -> ('α U)) => bool"
"correct_state σ ≡
(NULL /∈ dom σ ∧
(∀ obj ∈ ran σ. (level0.oclLib.is_OclAny_univ obj →
DEF(level0.oclLib.get_OclAny obj)) ∧
(∀ oid ∈ dom σ. level0.oclLib.is_OclAny_univ (the (σ oid)) →
OclOidOf0(level0.oclLib.get_OclAny (the (σ oid)))=oid)))"
```

The state is defined as a type definition using this invariant.

Operators: access, create, update

Large library of lemmas for the state.
SetColor

type: \textit{Cnode} \Rightarrow bool \Rightarrow state \Rightarrow state \ upward
SetColor

type: $Cnode \Rightarrow bool \Rightarrow state \Rightarrow state\ up$

consts $l1\_Cnode\_set\_color ::$
" $(('a, 'b) state, 'a Cnode) VAL$
$\Rightarrow (('a, 'b) state, bool up) VAL$
$\Rightarrow (('a, 'b) state$
$\Rightarrow (('a, 'b) state up)"
SetColor

defs l1_Cnode_set_Color_def :
" l1_Cnode_set_color self up_c σ ≡
(let oid = (l1_OclOid self ) σ in
let c0 = up_c σ in
let self0 = access_Cnode_in_state σ oid in
(if Cnode_in_state σ oid then
 updext_Cnode_in_state oid (base (Setcolor self0 c0)) σ )⊥
else ⊥))"
The commands of the language

Level 1 operators are:

- Setters
- update_news

They all fit in the Cmd slot provided by IMP++.

Instead `l1_Cnode_set_color N1 T σ` we write `N1 .color := T`, using syntax translation.

StackObject used for storing local variables.
The Program in IMP++

```plaintext
generate_cyclic_List  so ≡
so .n1 := New(Cnode);
so .n2 := New(Cnode);
so .cn1 .color := T;
(so .cn2) .color := F;
(so .n1) .next := (so .n2);
(so .n2) .next := (so .n1);
so .return := (so .n1)
```

Lukas Brügger

Proof Support for IMP++ in HOL-OCL
The Calculus

Large library of lemmas about what happens during a state transition:

- Definedness of expressions and objects
The Calculus

Large library of lemmas about what happens during a state transition:

- Definedness of expressions and objects

**lemma** DEF_set_color:

”\([\text{Cnode\_in\_state \(s\) (l1\_OclOid \text{self} \(s\))]} \implies \text{DEF} (l1\_\text{Cnode\_set\_color \text{self} \text{foo} \(s\)))”}
The Calculus

Large library of lemmas about what happens during a state transition:

- Definedness of expressions and objects
- Correct objects remain correct during an update
The Calculus

Large library of lemmas about what happens during a state transition:

- Definedness of expressions and objects
- Correct objects remain correct during an update

**lemma** Cnode_in_state_set_color:

"\[
\left[ \text{Cnode\_in\_state\ s oid ; l1\_OclOid self s = oid} \right] \implies \text{Cnode\_in\_state \neg l1\_Cnode\_set\_color self foo s} \neg \text{oid}"
"
The Calculus

Large library of lemmas about what happens during a state transition:

- Definedness of expressions and objects
- Correct objects remain correct during an update
- The value of a freshly set attribute
The Calculus

Large library of lemmas about what happens during a state transition:

- Definedness of expressions and objects
- Correct objects remain correct during an update
- The value of a freshly set attribute
- The value of attributes which didn’t get updated
The Calculus

Large library of lemmas about what happens during a state transition:

- Definedness of expressions and objects
- Correct objects remain correct during an update
- The value of a freshly set attribute
- The value of attributes which didn’t get updated
- Objects which won’t get updated remain the same
The Calculus

Large library of lemmas about what happens during a state transition:

- Definedness of expressions and objects
- Correct objects remain correct during an update
- The value of a freshly set attribute
- The value of attributes which didn’t get updated
- Objects which won’t get updated remain the same
- The free memory
The Calculus

Large library of lemmas about what happens during a state transition:

- Definedness of expressions and objects
- Correct objects remain correct during an update
- The value of a freshly set attribute
- The value of attributes which didn’t get updated
- Objects which won’t get updated remain the same
- The free memory
- Casting between class types
The Calculus

Large library of lemmas about what happens during a state transition:

- Definedness of expressions and objects
- Correct objects remain correct during an update
- The value of a freshly set attribute
- The value of attributes which didn’t get updated
- Objects which won’t get updated remain the same
- The free memory
- Casting between class types

The library is developed in a modular way. They are/will be created automatically by the encoder.
The Correctness Proof

\[
\models \lambda \sigma. \neg(\sigma \models \text{err}) \land \text{enough\_space\_for } \neg \sigma \downarrow 2 \land \\
\text{is\_handle\_for\_StackObject} \land \text{so \ so\_oid} \land \\
\text{StackObject\_in\_state} \land \neg \sigma \downarrow \text{so\_oid} \\
\text{so } .n1 := \text{New(Cnode)}; \\
\text{so } .n2 := \text{New(Cnode)}; \\
\text{so } .cn1 . color := T; \\
(\text{so } .cn2) . color := F; \\
(\text{so } .n1) . \text{next} := (\text{so } .n2); \\
(\text{so } .n2) . \text{next} := (\text{so } .n1); \\
\text{so } .\text{return} := (\text{so } .n1) \\
\{ \lambda \sigma. \neg(\sigma \models \text{err}) \land (\text{Cnode\_inv} (\text{so } .\text{return}) \neg \sigma \downarrow) \}\]

Lukas Brügger

Proof Support for IMP++ in HOL-OCL
The Correctness Proof

\[ \vdash \lambda \sigma. \neg (\sigma \models \text{err}) \land \text{enough_space_for} \ (\neg \sigma \downarrow 2 \land \text{is_handle_for_StackObject so so oid} \land \text{StackObject_in_state} \ (\neg \sigma \downarrow \text{so oid}) \land \text{generate_cyclic_list so} \]

\[ \lambda \sigma. \neg (\sigma \models \text{err}) \land (\text{Cnode_inv so . return} (\neg \sigma \downarrow)) \]"
The Correctness Proof

\[ \{\lambda \sigma. \neg(\sigma \models \text{err}) \land \text{enough\_space\_for} \ (\neg \sigma \downarrow 2 \land \text{is\_handle\_for\_StackObject} \ so \ so\_oid \land \text{StackObject\_in\_state} \ (\neg \sigma \downarrow so\_oid) \} \text{generate\_cyclic\_list} \ so \ \{\lambda \sigma. \neg(\sigma \models \text{err}) \land (\text{Cnode\_inv} \ (so \ . \ return) \ (\neg \sigma \downarrow))\} \]

- If we start in state which is not the error state, where there’s enough memory for two more objects, and where we have a StackObject of correct type, then
The Correctness Proof

\[ \{ \lambda \sigma. \neg (\sigma \models \text{err}) \land \text{enough_space_for } [\sigma] \land \text{is_handle_for_StackObject } \text{so so_oid } \land \text{StackObject_in_state } [\sigma] \text{ so oid} \} \]

\[ \{ \lambda \sigma. \neg (\sigma \models \text{err}) \land (\text{Cnode_inv } (\text{so . return}) [\sigma]) \} \]

- If we start in state which is not the error state, where there’s enough memory for two more objects, and where we have a StackObject of correct type, then

- after executing the method generate_cyclic_list on the StackObject,
The Correctness Proof

\[ \models \lambda \sigma. \neg (\sigma \models \text{err}) \land \text{enough\_space\_for} \ (\neg \sigma \downarrow 2 \land \text{is\_handle\_for\_StackObject} \ so \ so\_oid \land \text{StackObject\_in\_state} \ (\neg \sigma \downarrow \text{so\_oid}) \land \text{generate\_cyclic\_list} \ so \ \{ \lambda \sigma. \neg (\sigma \models \text{err}) \land (\text{Cnode\_inv} \ (\text{so} . \text{return}) \ (\neg \sigma \downarrow)) \} \]

- If we start in state which is not the error state, where there’s enough memory for two more objects, and where we have a StackObject of correct type, then
- after executing the method generate\_cyclic\_list on the StackObject,
- we end in a state which is not the error state and where the returned object satisfies the flip invariant.
Proof Outline

- A sequence of applications of rule semi_hoare, with explicit instantiations

\[
\{ A \} c \{ B \}; \quad \{ B \} d \{ C \} \\
\{ A \} c; d \{ C \}
\]
Proof Outline

- A sequence of applications of rule semi_hoare, with explicit instantiations
- Most lemmas of the library can safely be added to the simplifier, which proves most subgoals
- Only few direct rule applications necessary
- Final step a little more difficult. Through application of weak coinduction to several definedness expression and two inequalities.
Content

Introduction

Background

IMP++

Integrating IMP++ in HOL-OCL

Conclusion
Conclusion

- We showed the feasibility of a typed verification approach for object-oriented programs
- Large library of lemmas and definitions provides relatively efficient calculus for a Hoare logic
- Definition and lemmas are / will be created automatically by the encoder
- Tight integration with HOL-OCL allows taking advantage of the large library, and . . .
- . . . provides an integrated reasoning over object-oriented specifications and programs
Future Work

- Extend the encoder
- Support for method calls
- Verification Condition Generator